John von Neumann Institute for Computing

Efficient Object Placement including Node Selection in a Distributed Virtual Machine

Jose M. Velasco, David Atienza, Katzalin Olcoz, Francisco Tirado

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 509-516, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Efficient Object Placement including Node Selection in a Distributed Virtual Machine

Jose M. Velasco, David Atienza, Katzalin Olcoz, and Francisco Tirado

Computer Architecture and Automation Department (DACYA)
Universidad Complutense de Madrid (UCM)
Avda. Complutense s/n, 28040 - Madrid, Spain
E-mail: mvelascc@fis.ucm.es, {datienza, katzalin, ptirado}@dacya.ucm.es

Currently, software engineering is becoming even more complex due to distributed computing. In this new context, portability while providing the programmer with the single system image of a classical JVM is one of the key issues. Hence a cluster-aware Java Virtual Machine (JVM), which can transparently execute Java applications in a distributed fashion on the nodes of a cluster, is really desirable. This way multi-threaded server applications can take advantage of cluster resources without increasing their programming complexity.

However, such kind of JVM is not easy to design and one of the most challenging tasks is the development of an efficient, scalable and automatic dynamic memory manager. Inside this manager, one important module is the automatic recycling mechanism, i.e. Garbage Collector (GC). It is a module with very intensive processing demands that must concurrently run with user's application. Hence, it consumes a very critical portion of the total execution time spent inside JVM in uniprocessor systems, and its overhead increases in distributed GC because of the update of changing references in different nodes.

In this work we propose an object placement strategy based on the connectivity graph and executed by the garbage collector. Our results show that the choice of an efficient technique produces significant differences in both performance and inter-nodes messaging overhead. Moreover, our presented strategy improves performance with respect to state-of-the-art distributed JVM proposals.

1 Introduction

A cluster-aware Java Virtual Machine (JVM) can transparently execute Java applications in a distributed fashion on the nodes of a cluster while providing the programmer with the single system image of a classical JVM. This way multi-threaded server applications can take advantage of cluster resources without increasing programming complexity.

When a JVM is ported into a distributed environment, one of the most challenging tasks is the development of an efficient, scalable and fault-tolerant automatic dynamic memory manager. The automatic recycling of the memory blocks no longer used is one of the most attractive characteristics of Java for software engineers, as they do not need to worry about designing a correct dynamic memory management. This automatic process, very well-known as Garbage Collection/Collector (GC), makes much easier the development of complex parallel applications that include different modules and algorithms that need to be taken care of from the software engineering point of view. However, since the GC is an additional module with intensive processing demands that runs concurrently with the application itself, it always accounts for a critical portion of the total execution time spent inside the virtual machine in uniprocessor systems. As Plainfosse[5] outlined, distributed GC is even harder because of the difficult job to keep updated the changing references between address spaces of the different nodes.

Furthermore, the node choice policy for object emplacement is an additional task that can facilitate or difficult an efficient memory management. The inter-node message production increases proportionally to the distribution of objects that share dependencies. These dependencies can be seen as a connectivity graph, where objects are situated in the vertices and edges represent references.

In prior work, Fang et Al[11] have proposed a global object space design based on object access behaviour and object connectivity. Thus, they need to profile extensively the object behaviour. Their work is restricted to the allocation phase and it is not related to the GC. The main weakness of this approach is that knowledge of the connectivity graph during the allocation phase requires a lot of profile code and extra meta-data, which results in significant overhead in both performance and space.

In our proposal the object placement is based on object connectivity as well, but it is managed by the GC and takes place during its reclaiming phase. Hence, we have eliminated the profiling phase and the extra needed code. Moreover, after extensive experiments, we have observed that the tracing GC family is the optimal candidate to implement such a distributed scheme. Our results with distributed GCs show significant gains in performance and reductions in amount of exchanged data in the distributed JVM implementation in comparison to state-of-the-art distributed JVM schemes.

The rest of the paper is organized as follows. We first describe related work on both distributed GC and JVM. Then, we overview the main topics in distributed tracing GC. Next, we present our proposal based on suitable object placement and node selection during GC. Then, we describe the experimental setup used in our experiments and the results obtained. Finally, we present an overview of our main conclusions and outline possible future research lines.

2 Related Work

Different works have been performed in the area of both distributed garbage collection and distributed JVM. Plainfosse and Shapiro[5] published a complete survey of distributed GC techniques. In addition, Lins details a good overview of distributed JVMs within the Jones's classical book about GC[3].

A very relevant work in this area of distributed JVMs on a cluster of computing elements is the dJVM framework by Zigman et Al[8], which presents the distributed JVM as a single system image to the programmer. Although the approach is very interesting and novel, it is relatively conservative because it does not reclaim objects with direct or indirect global references. Our work in this paper is particularly enhancing this proposal (see Section 5.1) by removing its previously mentioned limitations to improve performance of the distributed JVM. Similar approaches with more limited scope of application to dJVM are cJVM[15] and Jessica[14]. cJVM, from Aridor et Al[7] is a distributed JVM built on top of a cluster enabled infrastructure, which includes a new object model and thread implementation that are hidden to the programmer. Thus, cJVM uses a master-proxy model for accessing remote objects. Jessica, from the university of Hong-Kong, employs a master-slave thread model. The negative part of these approaches is that for each distributed thread another thread exists in the master node, which handles I/O redirection and synchronization. Recently, Daley et Al[10] have developed a solution to avoid the need of a master node in certain situations.

Another variation to distributed JVMs is proposed by JavaParty[16] from the university of Karlsruhe. JavaParty uses language extensions to implement a remote method invocation, which is the main communication method in the cluster environment. The extensions are precompiled into pure Java code and finally into bytecode. However, Java language augmentation does not provide good solutions as it does not provide a true single system image.

Finally, in Jackal[12] and Hyperion[13] it is proposed a direct compilation of multi-threaded Java bytecode into C code, and subsequently into native machine code. Therefore, the Java runtime system is not used when executing applications. The main negative point of this scheme is the lack of flexibility and need to recompile the system when it is ported to another underlying hardware architecture executing the JVM.

3 Distributed Tracing Garbage Collection

In our work, we have developed a new framework for the analysis and optimization of tracing-based distributed GC by using the dJVM approach[8] as initial starting point. However, conversely to dJVM our new framework does not include any reference counting GC mechanisms[4], which are very popular in mono-processor GC solutions, because of two reasons. First, the reference counting GC is not complete and needs a periodical tracing phase to reclaim cycles, which will create an unaffordable overhead in execution time since it needs to block all the processing nodes. Second, the update of short-lived references produces continuous messages to go back and forth between the different nodes of the distributed GC, which makes this algorithm not scalable within a cluster.

On the contrary, as we have observed in our experiments, tracing GCs[4] seem to be the more convenient option to create such distributed GCs. Conceptually, all consist in two different phases. First, the marking phase allows the GC to identify living objects. This phase is global and implies scanning the whole distributed heap. Second, the reclaiming phase takes care of recycling the unmarked objects (i.e., garbage). The reclaiming phase is local to each node and can be implemented as a non-moving or as a moving GC. Thus, if objects are moved, we need to reserve space. The amount of reserved space must be equal to the amount of allocated memory. Hence, the available memory is reduced by two.

4 Object Placement During Garbage Collection

Our goal is to distribute objects in the cluster nodes according to a way that the communication message volume can be minimized. Ideally, a new object should be placed on the node where it is mainly required. Keeping this idea in mind, our proposed distributed scheme has a main node in which all new objects are allocated. Then, when this node runs out of memory, a local collection is triggered. During the tracing phase it is possible to know the exact connectivity graph. Therefore, we can select a global branch of the references graph and move to a node a complete group of connected objects.

It becomes apparent that our scheme incurs in a penalty, as objects are not distributed as soon as they are created. However, in the long run, this possible penalty is compensated by three factors. First, a high percentage of Java objects are very short-lived. Therefore, in our proposal, we have a higher probability of distributing long-lived data, which means that we

avoid segregating objects that may die quickly. Second, as our experimental results show (Section 5.3), we achieve a considerable reduction in the number of inter-node messages due to the interval before the object distribution phase is applied. Finally, since all the objects that survive a collection in the main node migrate to others nodes, we do not need to reserve space for them. Thus, in the main node we have all the possible available memory, without the reduction by a factor of two that occurs in all the moving GCs (see Section 3).

5 Experimental Setup and Results

In this section we first describe the whole simulation environment used to obtain detailed memory access profiling of the JVM (for both the application and the GC phase). It is based on cycle-accurate simulations of the original Java code of the applications under study. Then, we summarize the representative set of GCs used in our experiments. Finally, we introduce the sets of applications selected as case studies and indicate the main results obtained with our experiments.

5.1 Jikes RVM and dJVM

Jikes RVM is a high performance JVM designed for research. It is written in Java and the components of the virtual machine are Java objects[2], which are designed as a modular system to enable the possibility of modifying extensively the source code to implement different GC strategies, optimizing techniques, etc. There are different compiling options in Jikes. The baseline compiler does not perform any analysis and translates Java bytecodes to a native equivalent. In addition, Jikes RVM has an optimizing compiler and an adaptive compiler. Jikes is a Java virtual machine that runs on itself producing competitive performance with production JVMs.

The dJVM approach[8,9] is an enhancement of Jikes RVM to construct a distributed VM. Several infra-structured components were altered including inter-node communication, the booting process and the use of system libraries. It provides a single system image to Java applications and so it is transparent to the programmer. The dJVM employs a master-slave architecture, where one node is the master and the rest are slaves. The boot process starts at the master node. This node is also responsible for the setting up of the communication channels between the slaves, holding global data and the loading of application classes. The class loader runs in the master.

In dJVM objects are available remotely and objects have only a node local instance. This is achieved by using a global and a local addressing schemes for objects. The global data is also stored in the master with a copy of its global identifier in each slave node per object. Each node has a set of universal identifiers. Instances of primitives types, array types and most class types are always allocated locally. The exceptions are class types which implement the *Runnable* interface.

The initial design of dJVM targets the Jikes RVM baseline compiler. dJVM uses version 2.3.0 along with the Java memory manager Toolkit (JMTk)[1]. dJVM comes with different node selection policies: Round Robin (RR), random, etc. In our experiments, RR slightly outperforms the others. Thus, we use RR in the reported results is best policy choice for distributed JVMs. Finally, Jikes RVM code is scattered with a lot of assertion checking code that we have disabled for our experiments.

5.2 Case Studies

We have applied the proposed experimental setup to dJVM running the most representative benchmarks in the suite SPECjvm98 and the SPECjbb2000 benchmark[6]. These benchmarks were launched as dynamic services and extensively use dynamic data allocation. The used set of applications is the following:

_201_compress, _202_Jess, _209_db, _213_javac, _222_mpegaudio and _227_Jack. These benchmarks are not real multi-threading.

_228_mtrt: it is the multi-threaded version of _205_raytrace. It works in a graphical scene of a dinosaur. It has two threads, which make the render of the scene removed from a file of 340 KB.

The suite SPECjvm98 offers three input sets(referred as s1, s10, s100), with different data sizes. In this study we have used the biggest input data size, represented as s100, as it produces a bigger amount of cross-references among objects.

SPECjbb2000 is implemented as a Java program emulating a 3-tier system with emphasis on the middle tier. All three tiers are implemented within the same JVM. These tiers mimic a typical business application, where users in Tier 1 generate inputs that result in the execution of business logic in the middle tier (Tier 2), which calls to a database on the third tier. In SPECjbb2000, the user tier is implemented as random input selection.

We have also used a variant of the SPECjbb2000 benchmark for our experiments. SPECjbb2000 simulates a wholesale company whose execution consists of two stages. During startup, the main thread sets up and starts a number of warehouse threads. During steady state, the warehouse threads execute transactions against a database (represented as in-memory binary trees). This variant of SPECjbb2000 that we have used is called pseudojbb: pseudojbb runs for a fixed number of transactions (120,000) instead of a fixed amount of time.

5.3 Experimental Results

In our experiments we have utilized as hardware platform a 32-node cluster with a fast Ethernet communication hardware between the nodes. The networking protocol is standard TCP/IP. Then, each node is a Pentium IV, 866 MHz with 1024 Mb and Linux Red Hat 7.3.

Our experiments cover different types of configurations of the 32-node cluster, in the range of 2 to 32 processors. We have executed the benchmarks presented in the previous section in different single- and multi-threaded configurations to have a complete design exploration space. Precisely, we have executed pseudo JBB, _228_mtrt as multi-threaded applications, in combination with the rest of the other SPEC jvm98 and jbb2000 benchmarks, which are all single-threaded.

In our first set of experiments, shown in Fig. 1, we report the number of messages with distributed data that need to be exchanged between the nodes using our approach and in the case of the original dJVM. The results are normalized to the volume of messages of the original dJVM framework. These results indicate that our proposed approach reduces the number of messages that need to be exchanged with distributed information in all the possible configurations of processors with respect to dJVM, even in the case of single-threaded benchmarks. In fact, in complex and real-life multi-threaded applications, the reduction is very significant and always is above 20%. Moreover, the best configuration of distributed JVM is four processing nodes in both cases, and in this case the reduction in the

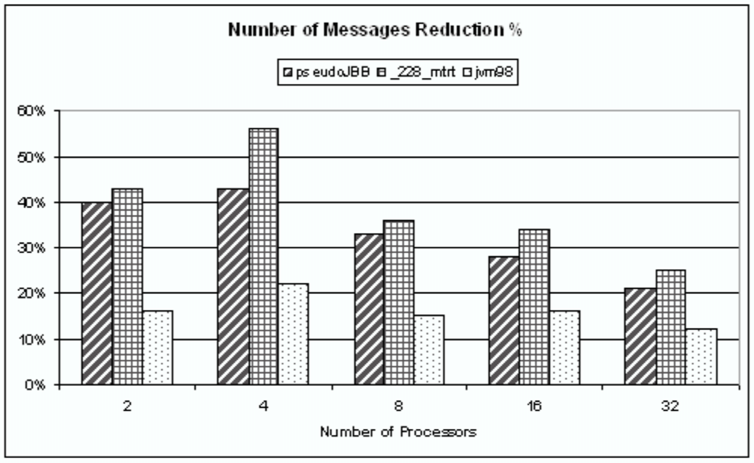

Figure 1. Reductions percentages in the number of messages exchanged between the 32 processing nodes of the cluster in our distributed JVM scheme for different sets of SPEC jvm98 and jbb2000 benchmarks. The results are normalized to the total amount of messages used in the dJVM

amount of exchanged data between nodes is approximately 40% on our side in comparison to dJVM.

In our second set of experiments, shown in Fig. 2, we have compared the execution time of our approach (node selection After Garbage Collection or AGC in Fig. 2) against Jikes RVM running on a uniprocessor system (or just one processor of the 32-node cluster). We have also compared the execution time of dJVM against Jikes RVM and the results are normalized in both cases to the single-processor Jikes RVM solutions. Thus, results in Fig. 2 greater than one mean that the distributed approaches are faster than the uniprocessor one.

Our results in Fig. 2 indicate that the previously observed reduction in the number of messages translates in a significant speed-up with respect to dJVM (up to 40% performance improvement) and single-processor Jikes RVM (up to 45% performance gains). In addition, in Fig. 2 we can observe that single-threaded applications (i.e., most of the jvm98 benchmarks) are not suited to distributed execution since the lack of multiple threads and continuous access to local memories severely affects the possible benefits of distributed JVM schemes. Thus, instead of achieving speed-ups, any distributed JVM suffers from performance penalties (up to 40%) with respect to monoprocessor systems, specially with a large number of processing nodes (i.e., 8 or more nodes) are used.

Finally, our results indicate that even in the case of multi-threaded Java applications of SPEC jvm98 and jbb2000 benchmarks, due to the limited number of threads available, the maximum benefits of distributed JVMs are with four processing nodes. Moreover, although pseudoJBB obtains forty percent speedups with eight processors, the difference in execution time related to four processors does not compensate for the amount of resources wasted in this computation and the observed computation efficiency of the overall cluster is lower in this case than with 4 processing nodes.

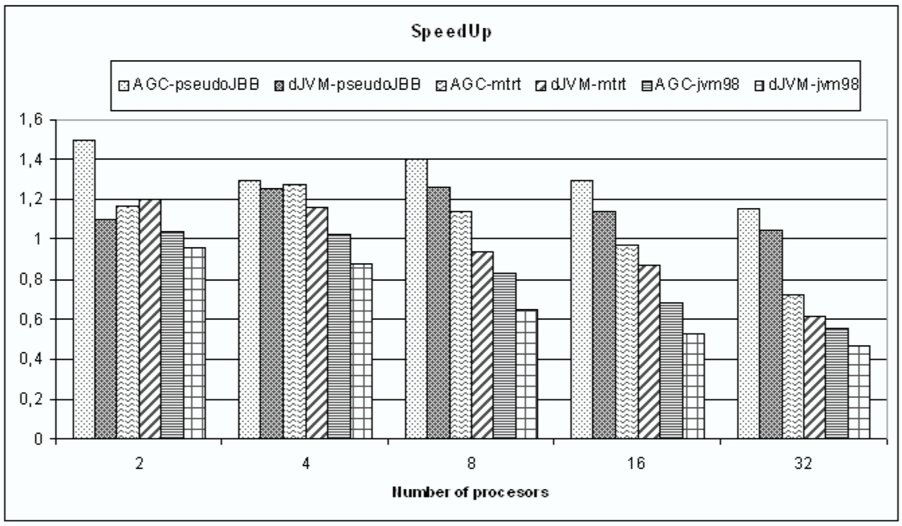

Figure 2. Performance speed-ups of different distributed JVMs for different SPEC jvm98 and jbb2000 benchmarks. The results are normalized to the Jikes RVM running locally in 1 cluster node

6 Conclusions and Future Work

In the last years software design has increased its complexity due to an extensive effort to exploit distributed computing. In this new working environment, portability while providing the programmer with the single system image of a classical JVM has become of the main challenges. Thus, it is really important to have efficient Java Virtual Machine (JVM) that can execute Java applications in a distributed fashion on the processing nodes of clusters, without incurring in significant processing efforts added to the software programmer. In this direction, JVMs need to evolve to provide an efficient and scalable automatic recycling mechanism or Garbage Collector (GC).

In this paper we have presented an object placement strategy based on the connectivity graph and executed by the GC. Our results show that our new scheme provides significant reductions in the amount of exchanged messages between the processing nodes of a 32-node cluster. Due to this reduction in the number of messages, our approach is faster than state-of-the-art distributed JVMs (up to 40% on average). In addition, our results indicate that distributed single-threaded applications (e.g., most benchmarks included in the SPEC jvm98) are not suited to distributed execution. Finally, our experiments indicate that the maximum gains are obtained with a limited number of processors, namely, 4 processors out of 32 in an homogeneous cluster.

In this work, our distributed JVM scheme places objects after the first GC in the master node occurs. As future work we intend to extend object migration after every global GC. Our belief is that that this feature can produce significant additional improvements in multi-threaded applications, both in execution time and number of messages reduction.

Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIN2005-05619.

References

1. IBM, *The Jikes' research virtual machine user's guide 2.2.0.*, (2003). http://oss.software.ibm.com/developerworks/oss/jikesrvm/
2. The source for Java technology, (2003). http://java.sun.com
3. R. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, 4th edition, (John Wiley & Sons, 2000).
4. R. Jones and R. D. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, (John Wiley & Sons, 1996).
5. D. Plainfosse and M. Shapiro, *A survey of distributed garbage collection techniques*, in: Proc. International Workshop on Memory Management, (1995).
6. SPEC, Spec documentation, (2000). http://www.spec.org/
7. M. Factor, Y. Aridor and A. Teperman, *A distributed implementation of a virtual machine for Java.* Concurrency and Computation: Practice and Experience.
8. J. Zigman and R. Sankaranarayanara, *djvm - a distributed jvm on a cluster*, Technical report, Australia University, (2002).
9. J. Zigman and R. Sankaranarayanara, *djvm - a distributed jvm on a cluster*, in: 17th European Simulation Multiconference, Nottingham, UK, (2003).
10. A. Daley, R. Sankaranarayana and J. Zigman, *Homeless Replicated Objects*, in: 2nd International Workshop on Object Systems and Software Architectures (WOSSA'2006), Victor Harbour, South Australia, (2006).
11. W. Fang, C.-L. Wang and F. C. M. Lau, *On the design of global object space for efficient multi-threading Java computing on clusters*, J. Parallel Computing, 11–12 Elsevier Science Publishers, (2003)
12. R. Veldema, R. Bhoedjang and H. Bal, *Distributed Shared Memory Management for Java*, Technical report, Vrije Universiteit Amsterdam, (1999).
13. The Hyperion system: Compiling multi-threaded Java bytecode for distributed execution. http://www4.wiwiss.fu-berlin.de/dblp/page/record/journals/pc/AntoniuBHMMN01
14. JESSICA2 (Java-Enabled Single-System-Image Computing Architecture version 2). http://i.cs.hku.hk/ clwang/projects/JESSICA2.html
15. Cluster Virtual Machine for Java. http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html
16. JavaParty. http://svn.ipd.uni-karlsruhe.de/trac/javaparty/wiki/